ELSEVIER

Science of Computer Programming 28 (1997) 1–41

Science of Computer Programming

The automated transformation of abstract specifications of numerical algorithms into efficient array processor implementations

Stephen Fitzpatrick [a,*], Terence J. Harmer [a], Alan Stewart [a], Maurice Clint [a], James M. Boyle [b,1]

[a] *Department of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, UK*
[b] *Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA*

Received 1 August 1995; revised 1 April 1996
Communicated by J. Darlington

Abstract

We present a set of program transformations which are applied automatically to convert abstract functional specifications of numerical algorithms into efficient implementations tailored to the AMT DAP array processor. The transformations are based upon a formal algebra of a functional array form, which provides a functional model of the array operations supported by the DAP programming language. The transformations are shown to be complete.

We present specifications and derivations of two example algorithms: an algorithm for computing eigensystems and an algorithm for solving systems of linear equations. For the former, we compare the execution performance of the implementation derived by transformation with the performance of an independent, manually constructed implementation; the efficiency of the derived implementation matches that of the manually constructed implementation.

Keywords: Program transformation; Program derivation; Normal forms; Functional specification; AMT DAP array processor

1. Introduction

The implementation of numerical mathematical algorithms on modern, high-performance computers presents an interesting contrast: most algorithms in this class have clear, easy-to-follow specifications, yet efficient implementations for high-performance computers are neither clear nor easy-to-follow.

* Corresponding author. E-mail: S.Fitzpatrick@cs.qub.ac.uk. Work supported by a research studentship from the Department of Education, Northern Ireland.

[1] This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, US Department of Energy, under Contract W-31-109-Eng-38.

PII S0167-6423(96)00015-9

That numerical mathematical algorithms have transparent specifications is not surprising – their mathematical foundation provides a coherent, logical and systematic framework and a rich body of knowledge that may be used to construct their specifications.

That acceptable implementations of numerical mathematical algorithms are rarely clear or easy-to-follow (or even correct!) is also not surprising – a programmer must usually formulate an implementation which differs radically from the specification in order to comply with the programming model supported by a particular implementation language and to exploit the specific hardware architecture in use (and thus improve the execution performance of the implementation). For example, a programmer may attempt to express certain operations in whole array or vector forms or to split a time-consuming task amongst a number of co-operating processes that execute concurrently on a parallel machine. Each implementation technique exacts a price as far as the clarity of the implemented program is concerned. When several implementation techniques are combined, the implementation becomes so complex that its relationship to the original algorithm specification is not apparent. Efficient implementations are thus often difficult to construct, to verify, to maintain and to adapt for execution on other computer systems.

In this paper, we discuss a method for *automatically deriving efficient implementations from abstract specifications through program transformation*. We distinguish two distinct roles in such a scheme:

(i) The *algorithm developer* constructs an abstract algorithm specification in a clear, natural style, paying no heed to efficiency.

(ii) The *transformational programmer* develops systematic implementation methods which are encoded as meaning-preserving program transformations.

The algorithm developer initiates the application of a sequence of program transformations, developed by the transformational programmer, to derive an efficient implementation (usually in Fortran or C) for the chosen computer system. The transformation sequence must implement the abstract constructs of the specification language in the target language, eliminate inefficiencies occasioned by the clear style of the specification and tailor the implementation for the chosen architecture.

Most transformations are independent of the particular algorithm being implemented, so a transformational programmer's efforts will be reused to produce implementations of other algorithms. In addition, many transformations are independent of the particular computer system for which an implementation is being derived, and can be reused for many computer systems. A single specification may serve as the source from which multiple implementations are derived, each implementation being tailored to a particular computer system.

The automated derivation of implementations for sequential and vector computer systems has been discussed previously [15]; in this paper, we extend this work to the derivation of implementations for the AMT DAP array processor. In Section 2 we discuss the specification language we use, a subset of the functional programming language ML [69], and the (small) set of functions that support array operations in ML and illustrate how these functions can be used to define common matrix and vector operations. In Section 3 we specify two significant algorithms that are used to

solve problems frequently encountered in scientific and engineering applications (the computation of eigensystems and the solution of systems of linear equations). We outline the AMT DAP architecture in Section 4 and then discuss the transformation system and the transformations used to produce DAP implementations in Section 5. Example applications of the transformations are given in Section 6 and an analysis of the execution performance of the derived implementations is given in Section 7. A discussion of related work and conclusions are presented in Sections 8 and 9.

2. A functional specification language for numerical mathematical algorithms

We use a (small) subset of the language constructs of ML as an algorithm specification language. We apply the term *functional specification* to an ML definition to convey that the definition is intended as an *abstract specification* of an algorithmic solution to a problem, *not* a *concrete program* to be executed in order to compute a solution efficiently. By regarding an ML definition as a specification, we liberate its style from all demands of efficient execution. Specifications can then be written in a style and using those techniques that produce the greatest degree of clarity, the strongest guarantee of correctness, and the greatest degree of adaptability. The problem of creating an executable, efficient, concrete implementation by automated program transformation is addressed later.

2.1. Vector and matrix primitives

Algorithms in numerical linear algebra are conventionally expressed in terms of operations on vectors and matrices, which we support through a library of standard operations, based upon an array data abstraction.

An array is defined as a mapping from a *Shape* to a set of values of a particular type: $array : Shape \rightarrow \alpha$. A *Shape* defines a set of indices (where an index is a list of integers specifying a position). We use the term *Shape* to emphasize that, in array operations, the set is usually *regular*; i.e. it can be specified using a small number of parameters.

In this paper, a *Shape* is defined by a number of dimensions with the details of each dimension expressed as a triple of the form: $[lower, upper, step]$ where *lower* is the smallest value in the set, *upper* is the largest value in the set and *step* is the offset between adjacent values. For example, a two-dimensional 4×4 *Shape* may be defined as $[[1,4,1],[1,4,1]]$ and denotes the set of indices $\{[i,j] | i \in 1..4 \wedge j \in 1..4\}$. For brevity, we use $[n]$ to denote a dimension with unit lower bound and offset; for example: $[n,n]$ is equivalent to $[[1,n,1],[1,n,1]]$.

The elements of a shape are indices, which are denoted as lists of values; for example, $[1,2]$ and $[4,1]$ are indices in the above 4×4 shape.

The library operations are defined in terms of three *primitive* functions for array element selection, array creation and array reduction.

Element Selection $element : \alpha\ array \times index \rightarrow \alpha$

Element selection is denoted using the function *element*; for example, $element(A, i)$ is (the value of) the element of array A at the position specified by index i. For convenience, an operator notation $A@i \equiv element(A, i)$ is also used.

Array Creation $generate : Shape \times (index \rightarrow \alpha) \rightarrow \alpha\ array$

An application of *generate* (called a *generation*) creates an array of the specified shape having elements given by applying the second argument (a function, called the *generating function*) to each index in the shape. Formally, *generate* is defined by:

$$v \in S \Rightarrow element(generate(S, \lambda x.E), v) \equiv \lambda x.E(v) \equiv E_v^x$$

where $\lambda x.E$ denotes a function with formal argument x and with body E, and where E_v^x denotes the result of substituting v for all free occurrences of x in the expression E. For example,

$$element(generate(S, \lambda[i,j].i + j), [1,2]) = (i+j)_{[1,2]}^{[i,j]} = 1 + 2 = 3.$$

Array Reduction $reduce : shape \times (index \rightarrow \alpha) \times (\alpha \times \alpha \rightarrow \alpha) \times \alpha \rightarrow \alpha$

A *reduction* combines a set of values into a cumulative value by means of a binary *reducing function* (the third argument to *reduce*). The set of values to be reduced is produced by applying a generating function (the second argument) to each index in a shape (the first argument). The final argument is the *initial value* – it is used to instantiate the reduction by inclusion in the set of values to be reduced (so that the application of a *binary* reducing function to the set is a valid operation even when the set contains only a single element). Formally, *reduce* can be defined by:

$$\begin{aligned} reduce(\{\}, \lambda x.E, op, a) &= a \\ reduce(S \cup \{y\}, \lambda x.E, op, a) &= reduce(S, \lambda x.E, op, a\ op\ E_y^x). \end{aligned}$$

No order for applying the generating or reducing functions is specified, so the reducing function should be associative and commutative.

Common examples of reductions are summing of the elements of a matrix, conjoining the elements in a boolean matrix, and determining the maximum value in a matrix.

(The *generate* and *reduce* functions have equivalent forms in a number of other programming languages and, in particular, the *C** and CM Fortran languages [22].)

The two functions *size* and *shape* are also used: $size(A, n)$ returns the size of A in the dimension specified by n; $shape(A)$ returns the shape of A.

The primitive array functions have been designed to provide a convenient means for defining common mathematical operations and to avoid biasing functions in favour of any particular computer architecture. For example, an application of *generate* or *reduce* can be evaluated either sequentially or in parallel – no order is specified for applying the generating function to the indices, or (in the case of *reduce*) for combining values.

Algebraic laws for the array operations are presented in Section 5.3.1.

2.2. Specifications for numerical mathematical algorithms

A library of standard matrix and vector operations is defined in terms of the primitive array operations. Most of the library operations are simple recastings of the conventional mathematical definitions. For example:

Matrix Addition $(A + B)[i,j] = A[i,j] + B[i,j]$

$$plus(A,\ B) = generate(shape(A),\ \lambda\ [i,\ j].A@[i,\ j]+B@[i,\ j])\ ^{2}$$

Matrix Transpose $A^{\mathrm{T}}[i,j] = A[j,i]$

$$transpose(A) = generate(shape\_transpose(shape(A)),\ \lambda\ [i,\ j].A@[j,\ i])$$
$$where\ shape\_transpose([n,\ m]) = [m,\ n].$$

Vector Inner Product $U.V = U[1] * V[1] + \cdots + U[n] * V[n]$

$$inner\_product(U,\ V) = reduce(shape(U),\ \lambda\ [i].U@[i]*V@[i],\ +,\ 0.0)$$

Matrix Multiplication $(A * B)[i,j] = row(A,i).column(B,j)$

$$multiply(A,\ B) = generate([size(A,\ 1),\ size(B,\ 2)],$$
$$\lambda\ [i,\ j].inner\_product(row(A,\ i),\ column(B,\ j)))$$

In each case the ML definition and the conventional mathematical form are closely related.

The definitions of commonly used functions such as *plus, transpose* and *multiply* have been placed in a library of numerical mathematical functions which are used in specifications. In most cases, the functions are invoked using standard operator notation – the specification language permits operators to be overloaded, so that, for example, "+" denotes matrix addition as well as integer addition and real addition.

The simplicity and clarity of functional programs make them particularly satisfactory for specifying numerical computations, especially when the basic specification language is enhanced by the inclusion of data abstractions. Data abstractions make it possible to introduce concepts and notations that are suited to the problem domain of a specification, or even to the particular problem under consideration.

3. Example specifications

In this section, we present specifications for two useful numerical mathematical algorithms as examples of more complex specifications.

[2] In this paper, λ-expressions are used to denote function expressions; ML uses the equivalent notation $fn(args) => expression$.

3.1. *An algorithm for computing eigensystems – Parallel Orthogonal Transformations (POT)*

The eigensystem (Q, Λ) of a matrix A of order n satisfies the equation $AQ = Q\Lambda$, where Λ is a diagonal matrix with the eigenvalues $\lambda_1, \ldots, \lambda_n$ of A as its diagonal elements, and the columns of Q are the corresponding eigenvectors. If A is symmetric, Q is guaranteed to be non-singular and is, in addition, orthogonal.

POT [20] computes the eigensystem of a symmetric matrix by constructing a sequence of orthonormal matrices of eigenvector approximations, $\{U_k\}$, and a sequence of similar symmetric matrices, $\{B_k\}$; thus:

(1) $U_0 = I$,
(2) $B_0 = A$,
(3) $B_k = U_k^{\mathrm{T}} A U_k$
(4) $U_{k+1} = ortho(AU_k transform(B_k), diagonal(B_k)), k \geqslant 0$.

Then $\lim_{k\to\infty}\{B_k\} = \Lambda$ and $\lim_{k\to\infty}\{U_k\} = Q$.

The function *transform* is defined below. The function *ortho* orthonormalizes the columns of its non-singular matrix argument using the modified Gram–Schmidt method. The columns of the argument matrix are orthogonalized in an order determined by the magnitude of the diagonal elements of B_k.

The eigenvectors and eigenvalues of a matrix A can be obtained by the ML definition

```
(eigenvectors, eigenvalue_matrix) = Pot(A, identity_matrix(shape(A)))
```

where the POT algorithm is realized as the ML function

```
fun Pot(A:real Array, U:real Array): real Array*real Array =
    let val B = transpose(U)*(A*U)
    in
        if (is_satisfactory(B))
        then (U, B)
        else Pot(A, ortho(A*U*transform(B), diagonal(B))
    end;
```

and where **let** ...**in** ...**end** defines a *local expression*: the identifier B is bound to the specified value (transpose...) during evaluation of the conditional expression; the value of the conditional expression is the value of the whole local expression.

This definition follows directly from the description of POT given above: if $B_k = U_k^{\mathrm{T}} A U_k$ is sufficiently close to being diagonal (as determined by the function is_satisfactory) then U_k is the matrix of eigenvectors, Q, and the diagonal elements of B_k are the required eigenvalues; otherwise a more accurate approximation to Q is derived and **Pot** is re-applied with this new approximation as its second argument.

The operation *transform* produces from its matrix argument a matrix T_k which, ignoring its diagonal, is anti-symmetric and each column of which is an approximation

$$t_{ij} = \begin{cases} \frac{2b_{ij}}{d_{ij}+sign(d_{ij})\sqrt{d_{ij}^2+4b_{ij}^2}}, & i > j \\ 1, & i = j \\ -t_{ji}, & i < j \end{cases}$$

where $d_{ij} = b_{jj} - b_{ii}$, and where b_{ij} is a typical element of B_k

(a) Mathematical definition [71]

```
fun transform(B:real Array):real Array =
let
    fun Calculate(i:int, j:int):real =
        let val d = B@[j, j]-B@[i, i]
        in 2*B@[i, j]/(d+sign(d)*sqrt(sqr(d)+4*sqr(B@[i, j])))
in
    generate(shape(B), λ [i, j]. if (i>j) then Calculate(i, j)
                                 else if (i=j) then 1.0
                                 else ~Calculate(j, i))
end
```

(b) ML specification

Fig. 1. The *transform* operation.

to an eigenvector of B_k. The components of T_k are computed as shown in Fig. 1(a). The ML specification shown in Fig. 1(b) defines a function **transform** that realizes the *transform* operation (the ML operator ~ denotes negation). This specification uses *generate* to construct the transformation matrix, T_k, which has the same shape as B_k. A function **Calculate** computes the value of the (i,j)th element of the transformation matrix. The generating function embodies the cases required by the specification. A similar development yields a specification for the function *ortho*.

3.2. A conjugate gradient algorithm

The conjugate gradient algorithm uses an iterative process to compute (an approximation to) the vector x of order n satisfying the equation $Ax = b$ where A is a positive-definite, symmetric matrix of order $n \times n$ and b is a vector of order n.

The name "Conjugate Gradient" often refers to a class of algorithms which employ the basic method defined in Fig. 2 [58, p. 152], rather than to a specific algorithm.

To solve $Ax = b$, where A is a positive-definite symmetric $n \times n$ matrix:

(i) Set an initial approximation vector x_0,
(ii) calculate the initial residual $r_0 = b - Ax_0$,
(iii) set the initial search direction $p_0 = r_0$;
(iv) then, for $i = 0, 1, \ldots,$
  (a) calculate the coefficient $\alpha_i = p_i^T r_i / p_i^T A p_i$,
  (b) set the new estimate $x_{i+1} = x_i + \alpha_i p_i$,
  (c) evaluate the new residual $r_{i+1} = r_i - \alpha_i A p_i$,
  (d) calculate the coefficient $\beta_i = -r_{i+1} A p_i / p_i^T A p_i$,
  (e) determine the new direction $p_{i+1} = r_{i+1} + \beta_i p_i$,
(v) continue until either r_i or p_i is zero.

Fig. 2. Mathematical definition of conjugate gradient.

The particular version used here is known as a *bi-conjugate gradient* algorithm; the functional specification is shown in Fig. 3.[3]

- The algorithm is based upon manipulation of a collection of vectors x, r, p and q (x being the current approximation to the solution); the type **cgstate** is defined to represent this collection of vectors, as a 4-tuple of real vectors. Instances of the **cgstate** type are constructed using the function **cgstate**.
- The function **cgiters** takes A and b as arguments and returns a **cgstate** whose first component is the solution.
- The specification uses the *iterate* library function to perform the repetition required by the algorithm.
  - The first argument to **iterate** is a function **cgiter** defining the computation that is to be repeated.
  - The second argument is a value (an instance of **cgstate**) with which to initialize the process.
  - The third argument, **has_converged**, is a function which determines when the repetition is to cease (i.e. when the approximation to the solution is sufficiently accurate).
- The function defining the repeated computation, **cgiter**, takes a single argument of type **cgstate** and returns a value of the same type. In the specification, pattern matching is used to bind the names **x**, **r**, **p** and **q** to the four components of the **cgstate** argument.
- The body of **cgiter** computes the next collection of vectors as local values **x'**, **r'**, **p'** and **q'** and returns these values as an instance of **cgstate**.
- For brevity, the computation of the initial values **x0**, **r0**, **p0** and **q0** is not shown.

The bulk of the computational costs are incurred by the two matrix–vector products in the computation of **atq** and **q'**.

[3] We emphasize that we are not interested in the merits and demerits of this particular algorithm – it is merely one that a *real* user was interested in and an example that was readily available to us.

```
type cgstate = real vector*real vector*real vector*real vector;
fun cgiters(a:real matrix, b:real vector):cgstate =
    let
        (* Terminating condition.*)
        fun has_converged((x, r, p, q):cgstate):bool =
            inner_product(r, r)<epsilon;
            (* One iteration.*)
        fun cgiter((x, r, p, q):cgstate):cgstate =
        let
            val rr:real = innerproduct(r, r);
            val alpha:real = rr/innerproduct(q, q);
            val x':real vector = x+p*alpha;
            val atq:real vector = transpose(a)*q;
            val r':real vector = r-atq*alpha;
            val beta:real = innerproduct(r', r')/rr;
            val p':real vector = r'+p*beta;
            val q':real vector = a*r'+q*beta
        in
            cgstate(x', r', p', q')
        end
    in
        iterate(cgiter, cgstate(x0, r0, p0, q0), has_converged)
    end
```

Fig. 3. SML specification of conjugate gradient.

The functional specifications presented above are straightforward recastings of the mathematical definitions into the chosen specification language. Although some of the syntactic detail differs from the mathematical form, the basic structures of the specifications mirror those of the mathematical definitions. The specifications should be readily understood by a reader with a knowledge of basic mathematics.

4. The target architecture: the AMT DAP 510

The AMT DAP 510 [59] is a Single Instruction Multiple Datastream (SIMD) parallel computer system, consisting of a 32 by 32 grid of processing elements (see Fig. 5) controlled by a separate master processor.

The master processor – essentially a conventional 32-bit processor with some additional components for controlling the operations of the processing elements – performs most scalar calculations. The processing elements, which are single-bit processors, perform the parallel processing operations. The master processor issues instructions to the processing elements, all of which obey the instruction simultaneously. The master processor may also issue data to the processing elements.

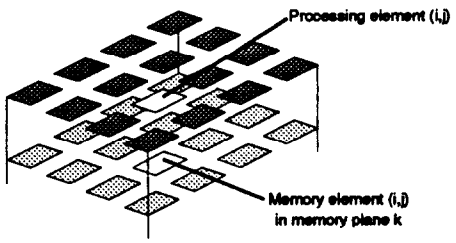

Fig. 4. DAP memory planes.

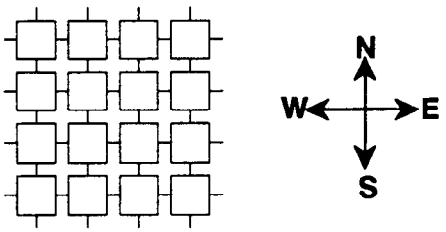

Fig. 5. DAP processor grid.

Each processing element has its own local memory to which it has direct access; no processing element has direct access to the memory of any other processing element. In general, in a given operation, all processing elements access the same component of their respective memories. Thus, the memory of all the processing elements may be thought of as consisting of a sequence of planes, the kth plane being the aggregate of the kth component of each processor's memory; the processor grid may be thought of as performing operations on these memory planes (see Fig. 4).

When a processing element requires a value which is stored in the memory of *another* element, it must obtain the value by a communication mechanism. Each processing element is connected to its four nearest neighbours in the grid, an element on an edge being connected to the corresponding element on the opposite edge (directions on the grid are designated as north, south, east and west – see Fig. 5); all of the processing elements can simultaneously obtain a value from one neighbour, though the direction in which each neighbour lies is the same across the entire grid.

In addition to the nearest neighbour connections, the DAP hardware supports three broadcast mechanisms which can be used to duplicate values across the grid: a single scalar value can be broadcast to each processing element, or a set of 32 scalar values (called a vector) can be broadcast to each row or to each column of the grid.

Associated with each processing element is an activity register which controls whether or not the element participates in certain operations. The activity mask (that is, the grid

of 32 by 32 activity registers) can be set under program control and can thus be used to implement conditional operations.

The DAP hardware also supports reduction operations (such as summation and conjunction) over the entire processor grid, and along only the rows or columns (to produce a vector of values).

4.1. The target language: Fortran Plus Enhanced

Fortran Plus Enhanced (FPE [1]) is an extension of standard Fortran allowing the processor grid of the AMT DAP to be used efficiently. It supports two non-conventional types, scalar vector and scalar matrix, which are similar to one-dimensional and two-dimensional arrays, but which have associated functions that make use of the processor grid.

The size of vectors or matrices which may be used is limited only by the amount of memory available, *not* by the size of the processor grid. Fortran Plus Enhanced subdivides a vector or matrix whose dimensions are larger than those of the processor grid into segments each of which is the size of the processor grid (if necessary, it pads the edges of the vector or matrix to make the size a multiple of the processor grid size).

The features of Fortran Plus Enhanced that are important in the context of this paper are:

Componental functions – scalar functions applied either to each element of a vector or matrix or to corresponding elements of a pair of vectors or matrices. The componental functions include common arithmetic and logical operations.
e.g. $A + B$, for vectors and matrices A and B.

Aggregate functions – certain elementwise reductions on a vector or matrix.
e.g. $sum(A)$, for a vector or matrix A.

Vector or matrix assignment – simultaneous assignment of all elements of a vector or matrix.
e.g. $A = B$, for vectors or matrices A and B.

Masked assignment – vector or matrix assignment controlled by a mask (a boolean vector or matrix).

Masked assignment affects only those elements of the left side vector or matrix for which the corresponding element of the mask is *true*.
e.g. $A(mask) = 1$, which assigns 1 to matrix A where the mask *mask* is *true*.

Masked vector or matrix assignment is the primary mechanism supporting conditional execution on the DAP processor array.

Pattern functions – construction of vector or matrix masks having *true* elements arranged in certain commonly used patterns.
For example: *patunitdiag*(N) is an $N \times N$ matrix with *true* along its leading diagonal and *false* everywhere else; *patlowertri*(N) is an $N \times N$ matrix with *true* in its lower triangle (the area on and below the leading diagonal) and *false* everywhere else.

Geometric functions – functions to re-arrange the order of elements in a vector or matrix.
e.g. *transpose*(A), for matrix A.

Extractions – a vector with elements equal to the elements of a given row or column of a matrix.
e.g. $A(1,)$ is row 1 of matrix A.

Complex extraction functions – extractions performed using a mask as an index.
For example, if M is a boolean matrix with one and only one element *true* in each row, then the positions of the *true* elements can be used to extract a vector from a matrix A, where A has the same size as a column of M. For example, *patunitdiag*(n) is a boolean matrix with *true* values along the main diagonal; $A(patunitdiag(n),)$ is a vector comprising the diagonal elements of A.

Expansion functions – a vector or matrix having each element equal to a given scalar value, or a matrix having each row or each column equal to a given vector.
e.g. $mat(1.0,m,n)$ is an $m \times n$ matrix with each element having the value 1.0, and $matr(V,m)$ is a matrix having m rows each of which is a copy of the vector V.

Shifts – vectors or matrices with all elements translated in the same direction. For example, a north shift moves all the elements of a matrix to the north, introducing null values along the south edge.

To run efficiently on the DAP, a program must be expressed almost entirely in terms of the operations described; operations which cannot be expressed as combinations of these operations are executed on the scalar processor, resulting in much slower execution than is achievable on the processor array.

5. Transforming functional specifications to efficient programs

The TAMPR program transformation system [10, 14] can be employed to apply program transformations to derive efficient Fortran or C programs from higher-order functional specifications. Each TAMPR transformation rule is a rewrite rule, having a pattern and a replacement, both of which are specified in terms of the grammar of wide spectrum language.

Most of the transformations that carry out such a derivation are independent of the problem being solved and of the target hardware, and so can be employed in derivations for any problem domain and for any target hardware architecture. As we discuss, however, one can easily add a few problem-domain-specific or hardware-specific transformations to the derivation to produce highly efficient code.

Typically, a derivation is structured into a sequence of major stages, each of which consists of a short sequence of transformation sets. TAMPR applies each transformation set once in turn, but exhaustively applies all transformations comprising that set. The total number of transformation applications may be large: for the POT specification, for example, the entire derivation from ML to Fortran Plus Enhanced requires about 150 000 rewrites. Clearly, it is vital that TAMPR supports the *automatic* application

of the rules. It would be unrealistic to attempt to apply thousands of transformations by hand, or even to guide their application.

5.1. Sketch of the basic transformational derivation

The stages in the basic transformational derivation are depicted in Fig. 6, in which the boxes represent particular transformation sequences and the arcs represent the order in which particular stages may be combined. The starting point for the derivation is a pure, functional specification (expressed in Lisp or ML); the result of the derivation is an imperative implementation expressed either in Fortran 77 or ANSI C.

The specification is transformed by:

(i) converting the specification into the abstract functional language used by the transformation system (essentially, the λ-calculus extended with named functions and type information);
(ii) standardizing the abstract functional language to facilitate later processing;
(iii) simplifying the structure of the functional specification by unfolding function definitions and evaluating certain resulting expressions;
(iv) converting the abstract functional form to an equivalent abstract imperative form; and
(v) converting the abstract imperative language to the required implementation language.

By removing the "syntactic sugar" of the initial specification (written in ML or in another functional language) the derivation is freed from the syntactic details of the functional language used for specification and thus permits other specification languages to be used with little additional effort.

Unfolding function definitions ensures that the only (non-recursive) functions persisting in a specification belong to a small set of designated "primitive" functions, such as *generate* and *reduce*. When defining the semantics of specifications or when transforming specifications, only the primitive functions need be considered after unfolding has been performed. The Unfolding and Simplification stage (stage 3) may be omitted.

The conversion of an abstract functional specification into an equivalent abstract imperative form (step 4) is achieved by:

(4.1) manipulating the functional specification into a form that renders the conversion to imperative form a straightforward task;
(4.2) performing tail recursion elimination on the abstract functional form;
(4.3) mapping the language constructs of the abstract functional language onto equivalent constructs in the abstract imperative language (for example, conditional expressions are mapped onto conditional statements); and
(4.4) implementing recursive functions by introducing a stack to store function arguments, return values and return addresses (thus removing the requirement on the implementation language to support recursive functions).

The tail recursion elimination and stack implementation phases may be omitted.

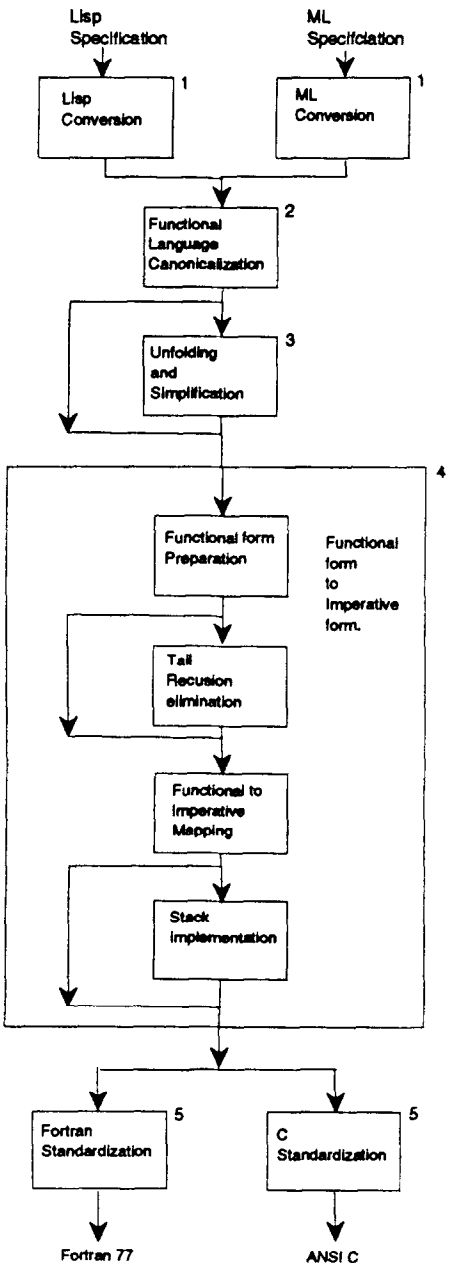

Fig. 6. Basic transformation derivation.

The transformations in the basic derivation provide the framework upon which other specialized derivations may be constructed. A more detailed discussion of the basic transformation steps, including some example code fragments generated at various stages, is given in [12, 15].

5.2. *Transformational derivation for the AMT DAP*

For efficient execution on the AMT DAP, a specification is recast into Fortran Plus Enhanced in order to exploit the parallel array operations provided on the processor grid. Rather than convert *directly* into FPE, the conversion is performed in two stages (see Fig. 7):

- In the first stage, array operations expressed in single-element terms are converted into whole-array operations. These whole-array operations are similar to those supported by Fortran Plus Enhanced, but they are all denoted as pure functions, whereas some of the FPE operations, such as masked array assignment, are destructive operations. The output form generated by this stage is called the Array Form.
- In the second stage, which augments the standard functional-to-imperative stage, the Array Form operations are converted into FPE operations.

Although it is based on operations supported by FPE, the Array Form is not intended to be DAP specific – the operations it supports are generic array-processor operations. The Array Form could thus be used as an intermediate form for *other* array processors, or for other implementation languages that are based on whole-array operations (such as Fortran90 or High Performance Fortran). Moreover, because the Array Form is a pure, functional form, it retains a simpler semantics than FPE, facilitating further manipulation such as common sub-expression elimination.

In addition to the two stages described above, a stage that uses algebraic properties of vectors and matrices to optimize specifications is included. For example, in the specification of POT, the expressions $U^{\mathrm{T}}AU$ and $AU\,transform(B)$ are evaluated. The matrix algebra stage ensures that the matrix product AU is computed only once, by rewriting these expressions as $U^{\mathrm{T}}(AU)$ and $(AU)transform(B)$. This optimization is obvious, and may seem trivial, but it has considerable effect on the execution performance (since the matrix product operation is so computationally expensive).

The array processor derivation outlined above may seem somewhat strange. The input to the derivation is an algorithmic specification that expresses the required operations in a form that permits data parallel implementation. This specification is reduced (we might say *simplified*) by unfolding to a form that consists only of *generate* and *reduce* functions. Thus, for example, an expression which may state array addition as A + B is transformed by function unfolding to

$$\text{generate (shape (A), } \lambda[i,j].\ A[i,j] + B[i,j])$$

where A and B are of the same array shape. We might be accused of converting a natural data parallel operation that is easily implementable on an array processor to one that must again be abstracted to A + B in our implementation.

Why is this appropriate? Firstly, not *all* data parallel operations in a specification will have natural implementations on an array processor; for example, consider the *transform* function above and its DAP Fortran Plus Enhanced implementation outlined

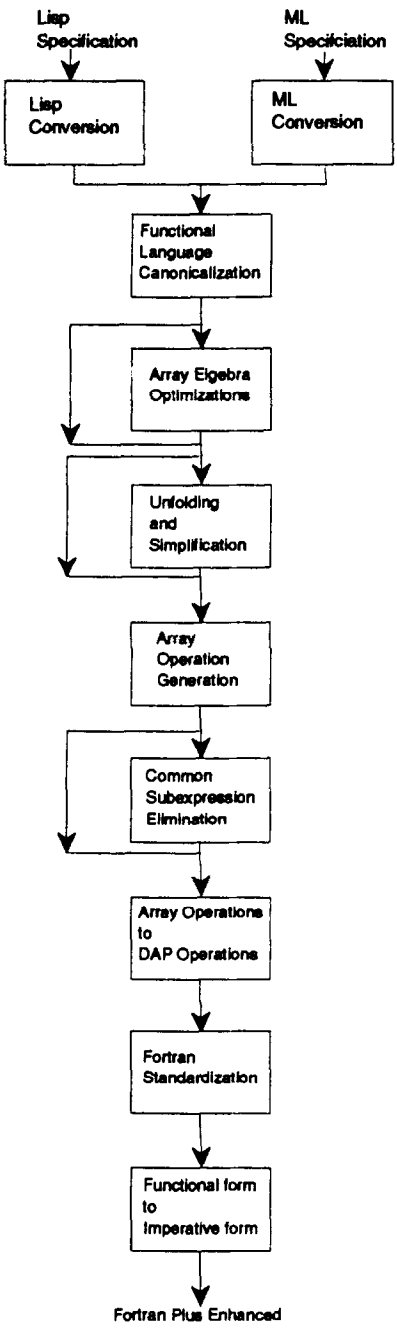

Fig. 7. AMT DAP transformation derivation.

later. More importantly, however, the specification may contain combinations of operations which when implemented directly do not give the best possible performance for a particular array processor. A simplified specification, consisting only of *generate* and *reduce* functions, permits the transformations to identify possible optimizations in a specification in a systematic manner without having to consider the many permutations of the possible data parallel operations.

5.3. Converting to Array Form

In this section we emphasize, in the main, the conversion from single-element to whole-array form; the conversion of the whole-array form operations into FPE operations is discussed briefly in a subsequent section.

The Array Form is based upon the λ-calculus augmented with a set of functions that perform generic array-processor operations. The additional functions correspond to the FPE operations discussed in Section 4.1. For example, the following operations are available:

- an operator $+_{array}$ for the elementwise addition of two arrays (the "array" subscript may be dropped in the discussion);
- a function *row* for extracting a specified row of a matrix;
- a function *sum* for summing the elements of a numeric array.

In addition, a data-parallel conditional expression, defined below, is used:

$$\begin{aligned} &\mathbf{If}_{array}\ M\ \mathbf{then}\ T\ \mathbf{else}\ F \\ &= generate(S, \lambda i.\mathbf{if}\ M@i\ \mathbf{then}\ T@i\ \mathbf{else}\ F@i) \end{aligned}$$

where M, T and F are arrays of shape S. The data-parallel conditional constructs an array by merging the elements of two arrays (T and F): a particular element of the result is drawn from T if the corresponding element of M (the "mask") is *true*; otherwise the element is drawn from F.

The purpose of the Array Form stage of the derivation is to convert array operations expressed using *generate* and *reduce* into Array Form operations. For example,

$$\begin{aligned} &generate([n,m], \lambda[i,j].A@[i,j] + B@[1,j]) \\ &\rightarrow A +_{array} expand\_rows(n, row(B,1)) \end{aligned}$$

where $expand\_rows(n, V)$ denotes a matrix having n rows, each of which is equal to the vector V. The advantage of the second, whole-array form is that it is easy to implement directly on an array processor. To implement directly and efficiently the first, single-element form on an array processor would be difficult.

The strategy that is used in the conversion to Array Form is to *simplify the internal structure of applications of generate by propagating generate inwards through arithmetic and other operations contained in generating functions*. For example,

$$
\begin{aligned}
& \mathit{generate}([n,m], \lambda[i,j].A@[i,j] + B@[1,j]) \\
\rightarrow\ & \mathit{generate}([n,m], \lambda[i,j].A@[i,j]) +_{array} \\
& \mathit{generate}([n,m], \lambda[i,j].B@[1,j]) \\
\rightarrow\ & A +_{array} \\
& \mathit{expand\_rows}(n, \mathit{generate}([m], \lambda[j].B@[1,j])) \\
\rightarrow\ & A +_{array} \mathit{expand\_rows}(n, \mathit{row}(B,1))
\end{aligned}
$$

- Each step of the transformation is based upon algebraic properties of *generate* and *reduce*, which are discussed below.
- Propagation through operators converts single-element operations into array operations.
- Propagation results in expressions such as

$$\mathit{generate}([n,m], \lambda[i,j].A@[i,j]) \text{ and } \mathit{generate}([m], \lambda[j].B@[1,j])$$

for which further propagation is impossible. The generating functions are assessed to determine whether or not they correspond to particular forms (such as "identity generate" or "row extraction"), which can be implemented efficiently on an array processor. Establishing such correspondences is facilitated by the simplified structure of the transformed generating functions (as compared with the structure of the original generating function).[4]

Below, the transformations that convert functional specifications into Array Form are discussed. The strategy used in applying these transformations ("propagation of *generate*") is described. Formal proofs that application of the transformations terminates under this strategy, and that application is complete, are given.

5.3.1. *Algebraic identities for generate and reduce*

The transformations that convert single-element form into Array Form are based upon algebraic identities for *generate* and *reduce*. These identities are listed here in three categories:

Propagation rules – which propagate applications of *generate* into expressions (thereby, for example, converting operators into whole-array form).

Special forms – which convert particular forms of *generate* into array operations (thereby, for example, extracting a row of a matrix).

Optimizations – which enhance the degree of parallelism in expressions. For example, converting multiple vector operations into a single matrix operation.

[4] Of course, not every residual generating function produced by propagation will correspond to an array processor operation. In such circumstances, efficient implementation on an array processor may not be possible.

Propagation rules

(i) **Infix element operator to infix array operator**

$$\begin{aligned} & generate(S, \lambda x.E_1 bop E_2) \\ \equiv\ & generate(S, \lambda x.E_1)\ bop_{array}\ generate(S, \lambda x.E_2) \end{aligned}$$

where *bop* is a binary infix operator.

A generation constructed from the expression E_1 *bop* E_2 is equivalent to the array version of *bop* applied to arrays constructed from E_1 and E_2. For example,

$$\begin{aligned} & generate(S, \lambda x.E_1 + E_2) \\ \equiv\ & generate(S, \lambda x.E_1) +_{array} generate(S, \lambda x.E_2) \end{aligned}$$

(ii) **Unary element operator to unary array operator**

$$generate(S, \lambda x.uopE) \equiv uop_{array}\ generate(S, \lambda x.E)$$

where *uop* is a unary operator.

An array constructed from *uopE* is equivalent to the elementwise application of *uop* to the array constructed from expression E. For example,

$$generate(S, \lambda x.\mathbf{abs}E) \equiv \mathbf{abs}_{array} generate(S, \lambda x.E)\ .$$

(iii) λ **promotion from** *generate*

$$\begin{aligned} & generate(S, \lambda x.((\lambda y.E1)E2)) \\ \equiv\ & ((\lambda Z.generate(S, \lambda x.E1^{y}_{Z[x]}))generate(S, \lambda x.E2)) \end{aligned}$$

Consider the left side of this identity: an array is constructed in which each element requires the evaluation of expression $E2$ and the binding of the result to identifier y. There is no mechanism in FPE for constructing such an array in parallel; the construction would have to be implemented in FPE as a sequential loop.

However, the binding of $E2$ to y can, potentially, be performed for all elements in parallel by constructing a separate array, Z, as shown on the right of the identity – the value of $E2$ for each index x is stored as an element of Z. The original array is constructed as before except that the binding for y is replaced with an access to the appropriate element of Z. It may then be possible to construct each array using whole-array operations. For example,

$$\begin{aligned} & generate(S, \lambda x.(\lambda y.(y + sqrt(y))(2 * A[x]))) \\ \equiv\ & (\lambda Z.generate(S, \lambda x.(y + sqrt(y))^{y}_{Z[x]})) \\ & (generate(S, \lambda x.2 * A[x])) \\ \equiv\ & (\lambda Z.generate(S, \lambda x.Z[x] + sqrt(Z[x]))) \\ & (generate(S, \lambda x.2 * A[x])) \\ \equiv\ & (\lambda Z.Z +_{array} sqrt_{array}(Z))(2 *_{array} A) \end{aligned}$$

(iv) **Conditional expressions**

$$\begin{aligned} & \mathit{generate}(S, \lambda x.\textbf{if } E_b \textbf{ then } E_1 \textbf{ else } E_2) \\ \equiv\ & \textbf{if}_{\mathit{array}}\ \mathit{generate}(S, \lambda x.E_b) \textbf{ then } \mathit{generate}(S, \lambda x.E_1) \\ & \textbf{else } \mathit{generate}(S, \lambda x.E_2) \end{aligned}$$

This identity is essentially the definition of the data-parallel conditional.

Special forms

Array processor programming languages generally include a predefined set of optimized methods for performing certain operations – primarily communication operations – commonly required for numerical mathematical algorithms. To produce an efficient program, these standard optimizations *must* be exploited. Thus, it is necessary to identify, from the array expressions within a specification, those expressions that are instances of supported operations. Identifying such expressions is facilitated by the simplification of generating functions that results from the propagation of *generate* carried out by the preceding set of transformations.

(v) **Array identity**

$$\mathit{generate}(S, \lambda x.A[x]) \equiv A \text{ where } \mathit{Shape}(A) = S.$$

(vi) **Array constants**

$$\mathit{generate}(S, \lambda x.e) \equiv \mathit{expand}(S, e)$$

where e is independent of the generating index x and $\mathit{expand}(S, e)$ is an array of shape S with each of its elements having the value e. For example,

$$\mathit{generate}([n], \lambda x.1.0) \equiv \mathit{expand}([n], 1.0)$$

is a vector of length n with each element having the value 1.0.

(vii) **Column or row expansion**

$$\begin{aligned} & \mathit{generate}([n, m], \lambda[i, j].E) \\ \equiv\ & \mathit{expand\_cols}([m], \mathit{generate}([n], \lambda[i].E)) \end{aligned}$$

where E is independent of j, and

$$\begin{aligned} & \mathit{generate}([n, m], \lambda[i, j].E) \\ \equiv\ & \mathit{expand\_rows}([n], \mathit{generate}([m], \lambda[j].E)) \end{aligned}$$

where E is independent of i.

Constructing a matrix by applying a generating function that is independent of one of the indices is equivalent to constructing a vector and duplicating the vector row- or column-wise, as appropriate.

(viii) **Array patterns**

$$\begin{aligned} & \mathit{generate}([n, n], \lambda[i, j].i = j) \\ \equiv\ & \mathit{generate}([n, n], \lambda[i, j].i) =_{\mathit{array}} \mathit{generate}([n, n], \lambda[i, j].j) \\ \equiv\ & \mathit{diagonal\_pattern}(n) \end{aligned}$$

where *diagonal_pattern*(n) is an $n \times n$ boolean matrix with each of its diagonal elements having the value *true*, and all of its other elements having the value *false*. Identities exist for other patterns, corresponding to other comparison operators such as 'less than'.

(ix) **Permutations**

$$generate([n,m], \lambda[i,j].A[j,i]) \equiv transpose(A), \quad shape(A) = [m,n]$$

Transpose is the most common permutation.

(x) **Extractions**

$$\begin{aligned} generate([n], \lambda[i].A[i,i]) &\equiv diagonal(A), & shape(A) &= [n,n] \\ generate([n], \lambda[i].A[i,k]) &\equiv column(A,k), & shape(A) &= [n,m] \\ generate([m], \lambda[i].A[k,i]) &\equiv row(A,k), & shape(A) &= [n,m] \end{aligned}$$

where, in each case, k is independent of i.

(xi) **Shifts**

$$\begin{aligned} & generate([n,m], \lambda[i,j].\textbf{if } i = 1 \textbf{ then } 0 \textbf{ else } A[i-1,j]) \\ \equiv\ & \textbf{If}_{array}\ generate([n,m], \lambda[i,j].i = 1) \\ & \textbf{then } generate([n,m], \lambda[i,j].0) \\ & \textbf{else } generate([n,m], \lambda[i,j].A[i-1,j]) \\ \equiv\ & ShiftSouth(A) \end{aligned}$$

where $shape(A) = [n,m]$.

An expression of the first form is converted into an expression of the second form by the propagation rules and is then converted into an application of *ShiftSouth*. Similar rules apply for shifts in other directions, for combinations of shifts (such as a north-east shift) and for unidirectional shifts of magnitude greater than 1.

Optimizations

To obtain optimum performance from a DAP implementation, it is necessary to augment the preceding rules with others that are designed to take advantage of the particular capabilities of an array processor architecture and, in particular, those of the AMT DAP.

For an array processor architecture, it is preferable that a single large data-parallel operation be performed rather than a sequence of smaller data-parallel operations – this means that it is worthwhile to seek to combine or reorder sequences of *generate* and *reduce* operations to give a data-parallel operation that applies to the largest possible number of array elements.

(xii) *reduce-reduce* **combination**

$$\begin{aligned} & reduce(S, \lambda[x].reduce(S', \lambda[y].E, \odot, 0_\odot), \odot, 0_\odot) \\ \equiv\ & reduce(S \times S', \lambda[x,y].E, \odot, 0_\odot) \end{aligned}$$

where S' is independent of x, $\times$ denotes the cartesian product of shapes, and $0_\odot$ is an identity element of operator $\odot$.

This identity asserts that a reduction, using an operator $\odot$, of a set of values each of which is itself the result of a reduction using $\odot$, is equivalent to a single reduction. For example,

$$\begin{aligned} & \mathit{reduce}([n], \lambda[x].\mathit{reduce}([m], \lambda[y].A[x, y], +, 0), +, 0) \\ \equiv\ & \mathit{reduce}([n, m], \lambda[x, y].A[x, y], +, 0) \end{aligned}$$

This optimization establishes a larger parallel reduction from a number of smaller reductions – by converting, in this example, $n + 1$ vector reductions into a single matrix reduction.

This rule can be generalized to reductions in which the initial value is *not* the identity element of the reducing function.

(xiii) *generate-reduce* **swap**

$$\begin{aligned} & \mathit{generate}(S_1, \lambda[x, y].\mathit{reduce}(S_2, \lambda[z].E, \odot, v)) \\ \equiv\ & \mathit{reduce}(S_2, \lambda[z].\mathit{generate}(S_1, \lambda[x, y].E), \odot_{\mathit{array}}, v_{\mathit{array}}) \end{aligned}$$

where S_2, v are independent of x, and where $\odot_{\mathit{array}}$ and v_{array} on the right side are an array operator and an array of initial values, respectively ($v_{\mathit{array}} \equiv \mathit{expand}(S_1, v)$).

This identity asserts that the evaluation of multiple reductions, each of which produces a single element of a matrix, is equivalent to a single reduction which constructs the complete matrix, using the array version of the reducing function. This optimization is important in the context of the matrix product operation:

$$\begin{aligned} & \mathit{generate}([n, m], \lambda[x, y].\mathit{reduce}([l], \lambda[z].A@[x, z] * B@[z, y], +, 0)) \\ \equiv\ & \mathit{reduce}([l], \lambda[z].\mathit{generate}([n, m], \lambda[x, y].A@[x, z] * B@[z, y]), +, 0) \end{aligned}$$

The left side corresponds to the *ijk* order of evaluation (with k parallelised); the right side corresponds to the *kij* order of evaluation (with *ij* parallelised). As discussed in [16], the latter order of evaluation can be understood as computing the matrix product by a sequence of n rank-one updates to the zero matrix.

The motivation for this optimization is as follows: the *reduce-generate* combination can be considered as exhibiting three-dimensional parallelism (the expression E must be evaluated for each combination of x, y and z, in the appropriate ranges). However, the DAP can utilize at most two-dimensional parallelism, so that at least one dimension must be processed sequentially. Because reductions tend to exploit parallelism less than other operations (such as elementwise operations)[5], it is an optimization to use this identity to arrange for the reduction to be the operation that is performed sequentially.

(xiv) *generate-reduce* **combination**

$$\begin{aligned} & \mathit{generate}([m], \lambda[x].\mathit{reduce}([n], \lambda[y].E, \odot, v)) \\ \equiv\ & \mathit{reduce\_rows}(\mathit{generate}([m, n], \lambda[x, y].E), \odot, v) \end{aligned}$$

where n and v are independent of x and *reduce_rows* reduces its matrix argument along its rows to form a vector of values.

[5] The parallel reduction of a vector of length n by an array processor typically requires $\log_2(n)$ steps, whereas the parallel addition of two vectors of length n can be performed in one step.

This identity asserts that the evaluation of multiple reductions, each of which creates a single element of a vector, is equivalent to the construction of a matrix followed by a reduction along its rows.[6] This optimization improves performance by increasing the degree of parallelism – if the *generate* and *reduce* were *not* combined, the *generate* would be evaluated sequentially. For example, matrix–vector product is optimized as:

$$\begin{aligned} & generate([m], \lambda[x].reduce([n], \lambda[y].A@[x,y] * U@[y], +, 0)) \\ \equiv\ & reduce\_rows(generate([m,n], \lambda[x,y].A@[x,y] * U@[y]), +, 0) \end{aligned}$$

(xv) *generate-generate* **combination**

$$generate(S, \lambda x.generate(S', \lambda y.E)) \equiv generate(S \times S', \lambda x \times y.E)$$

This identity asserts that an array of arrays is considered equivalent to a single, "flattened" array. This equivalence is included for completeness; it is not used in practice since it requires a more complex interpretation of basic operations. For example, array indexing must be "curried" so that, say, a two-dimensional index applied to a four-dimensional array returns a two-dimensional array.

5.3.2. *Transformation application strategy*

The equivalences in Section 5.3.1, when used left-to-right, constitute the transformations required when converting an abstract functional specification into an efficient form suitable for execution on an array processor architecture and, in particular, on the AMT DAP. The rules involve patterns that are disjoint; thus, no transformation interferes with any other transformation (i.e., for a given program section, at most one transformation is applicable). In addition, the rules cannot result in an infinite sequence of transformations (see the next section). Thus, the rules can be applied *automatically* to transform an element-based functional specification to an array-based specification optimized for an array processor architecture.

5.3.3. *Completeness proof and normal form*

The DAP transformation strategy propagates *generate* functions into expressions as far as is possible. This strategy may be viewed as a way of deriving a normal form for *generate* and *reduce*, since these functions cannot be driven indefinitely far into expressions. The existence of a normal form enables the transformations to be applied automatically in the TAMPR system, without the need for human guidance.

The basic idea is illustrated by demonstrating how a *generate* term may be transformed into Array Form. The individual rewrites used in the transformation process are equivalences in the algebra of *generate* – see Section 5.3.1 – just as the rewrites used in earlier transformation stages are equivalences in the λ-calculus.

For simplicity, the discussion concentrates on the propagation rules, and it is assumed that the elements of arrays are scalar values (integers, reals or booleans) and assumed

[6] A column-wise combination is more efficient than a row-wise combination for some expressions. The transformations used in practice include heuristics to decide which to use.

to have the form $generate(S, \lambda x.E)$, where E is defined (using Extended Backus Naur Form) as

$$E :: C|V|N(E)|uop\ E|E_1 bop E_2|\textbf{if}\ E_b\ \textbf{then}\ E_1\ \textbf{else}\ E_2\ |((\lambda y.E_1)E_2)$$

where y denotes a tuple of names; C denotes a constant; V denotes a λ variable; E and E_i denote expressions; and $N(E)$ denotes a function application. The details of the classes C, V and N are irrelevant in the derivation of the DAP normal form.

The application of the transformations can be represented by a recursive tactic T, defined by:

$$\begin{array}{lllll}
T(generate(S,\lambda x.E)) \stackrel{\text{def}}{=} & & & & \\
\quad case\ E\ of & & & & \\
\qquad C & & \rightarrow & generate(S,\lambda x.C) & \\
\qquad V & & \rightarrow & generate(S,\lambda x.V) & \\
\qquad N(E') & & \rightarrow & generate(S,\lambda x.N(E')) & \\
\text{(Rule i)} & E_1\ bop\ E_2 & \rightarrow & T(generate(S,\lambda x.E_1))\ bop & \\
 & & & T(generate(S,\lambda x.E_2)) & \\
\text{(Rule ii)} & uop\ E' & \rightarrow & uop\ T(generate(S,\lambda x.E')) & \\
\text{(Rule iii)} & ((\lambda y.E1)E2) & \rightarrow & (\lambda Z.T(generate(S,\lambda x.E1^{y}_{Z(x)}))) & \\
 & & & T(generate(S,\lambda x.E2)) & \\
\text{(Rule iv)} & \textbf{if}\ E_b & & & \\
 & \textbf{then}\ E_1 & & & \\
 & \textbf{else}\ E_2 & \rightarrow & \textbf{if}\ \ (T(generate(S,\lambda x.E_b))) & \\
 & & & \textbf{then}\ \ T(generate(S,\lambda x.E_1)) & \\
 & & & \textbf{else}\ \ T(generate(S,\lambda x.E_2)) &
\end{array}$$

It is important to note that unary and binary operators (syntactic classes *uop* and *bop*) are overloaded: on the left of the rewrites they are applied to individual elements while on the right they are applied to structures.

Proposition 1. *After transformation, all remaining generate terms have the form tg defined by*

$$tg :: generate(S,\lambda x.C)|generate(S,\lambda x.V)\ |generate(S,\lambda x.N(E))$$

Proof. Define a measure μ on generating functions; this measure induces an ordering which is used to establish Proposition 1 by structural induction. The measure also facilitates a proof of termination of the transformation.

The definition of μ is:

$$\begin{array}{rl}
\mu(generate(S,E)) \stackrel{\text{def}}{=} & case\ E\ of \\
C & = 1 \\
V & = 1 \\
N(E) & = 1
\end{array}$$

$$
\begin{aligned}
uop\ E &= 1 + \mu(E) \\
E_1\ bop\ E_2 &= 1 + \mu(E_1) + \mu(E_2) \\
\textbf{if}\ E_b\ \textbf{then}\ E_1\ \textbf{else}\ E_2 &= 1 + \mu(E_b) + \mu(E_1) + \mu(E_2) \\
((\lambda y.E_1)E_2) &= 1 + \mu(E_1) + \mu(E_2)
\end{aligned}
$$

The relation $<_\mu$ on generating functions is defined as $E <_\mu E' \equiv \mu(E) < \mu(E')$, where $<$ is the usual "less than" relation on natural numbers. The salient point of the ordering on generating functions is that "compound" expressions (unary and binary operator expressions, conditional expressions and λ-applications) are "larger" than their constituent sub-expressions.

Let $v(E)$ denote the property that all generations occurring in E have the form tg. It is shown that:

- $v(T(generate(S, \lambda x.e)))$ holds for base cases of E (viz. C, V and $N(E)$), and that
- if $v(T(generate(S, \lambda x.e')))$ holds for all $e' <_\mu e$, then $v(T(generate(S, \lambda x.e)))$ also holds.

Then, by structural induction, $v(T(generate(S, \lambda x.e)))$ holds for all e.

Base steps: Consider case C of E. A generation with generating function of this form is left unchanged by T. It is already in the form tg, so $v(T(generate(S, \lambda x.C)))$ holds. Similarly for the cases V and $N(E)$.

Inductive steps: Consider the case $E = \textbf{if}\ E_b\ \textbf{then}\ E_1\ \textbf{else}\ E_2$.

$$
\begin{aligned}
& v(T(generate(S, \lambda x.\textbf{if}\ E_b\ \textbf{then}\ E_1\ \textbf{else}\ E_2))) \\
= {} & v(\textbf{if}\ T(generate(S, \lambda x.E_b)) \\
& \textbf{then}\ T(generate(S, \lambda x.E_1))\ \textbf{else}\ T(generate(S, \lambda x.E_2))) \\
= {} & v(T(generate(S, \lambda x.E_b))) \wedge v(T(generate(S, \lambda x.E_1))) \\
& \wedge v(T(generate(S, \lambda x.E_2))) \\
= {} & true \wedge true \wedge true \text{ by hypothesis, since } E_b, E_1, E_2 <_\mu E \\
= {} & true
\end{aligned}
$$

The cases for unary and binary operators follow similarly. The case

$$E = ((\lambda x.E_1)E_2)$$

requires a little more attention:

$$
\begin{aligned}
& v(T(generate(S, \lambda x.((\lambda y.E_1)E_2)))) \\
= {} & v(((\lambda Z.T(generate(S, \lambda x.(E_1)^y_{Z[x]})))\ T(generate(S, \lambda x.E_2)))) \\
= {} & v(T(generate(S, \lambda x.(E_1)^y_{Z[x]}))) \wedge v(T(generate(S, \lambda x.E_2)))
\end{aligned}
$$

Now the second term in the above, $v(T(generate(S, \lambda x.E_2)))$, holds by the induction hypothesis, since $E_2 <_\mu E$. Since $\mu(y) = \mu(Z[x])$, the substitution of the latter for the former in an expression leaves the measure of the expression exchanged: that is, $\mu((E_1)^y_{Z[x]}) = \mu(E_1)$. Now $E_1 <_\mu E$, so $(E_1)^y_{Z[x]} <_\mu E$ and $v(T(generate(S, \lambda x.(E_1)^y_{Z[x]})))$ follows by the induction hypothesis.

Thus, by structural induction, Proposition 1 holds.

Corollary. *It is possible to detect, in the normal form, generate (and reduce) terms which have data-parallel implementations using rules v–xi.*

5.4. Converting array form operations into FPE

Many of the Array Form functions have direct equivalents in Fortran Plus Enhanced. For example, whole-array operators such as $+_{array}$ map onto whole-array versions of standard operators (+); row and column extraction map onto special indexing forms (e.g. $row(A,i) \rightarrow A(i,))$; scalar and vector expansions map onto FPE functions (e.g. $expand([n,m],e) \rightarrow mat(e,n,m)$).

Data-parallel conditional expressions are realized as masked assignments. For example, the expression $\mathbf{if}_{array}\ E_b$ **then** E_1 **else** E_2 maps onto the sequence of array assignments:

```
mask = E_b
A(mask) = E_1
A(.not.mask) = E_2
```

(There are also various optimized implementations for certains forms of data-parallel conditionals; for example, updating a single element of a matrix can be implemented as a standard, single-element assignment without using a mask.)

Reductions that have not been converted into Array Form primitives (such as *sum*) are implemented in FPE as loops. For example, $reduce([n], \lambda[k].E, +, 0)$ is converted into

```
result = 0
DO k=1, n, 1
  result = result+E
ENDDO
```

6. DAP implementations

To illustrate the use of the transformations discussed above, we consider in detail the transformation of part of the function **Pot**, whose specification is discussed in Section 3.1. The derived implementation of CG is also discussed briefly.

6.1. POT

(i) **Functional language standardization**

```
fun Pot:real Array =
  λ A:real Array.
    λ U:real Array.
      λ B:real Array.
        if (is_satisfactory(B))
```

```
            then (U, B)
            else Pot(A, ortho(mmmult(A, mmmult(U, transform(B))),
                              diagonal(B)))
          end (mmmult(transpose(U), mmmult(A, U)))
        end
      end
    end
```

Infix operators have been converted to applications of functional equivalents and λ-bindings have been introduced for ML **let** bindings.

(ii) **Matrix algebra optimizations**

The repeated calculation of matrix $A \times U$ is recognized and bound to the name **AU** to ensure it is evaluated only once.

```
fun Pot:real Array =
  λ A:real Array.
    λ U:real Array.
      λ AU:real Array.
        λ B:real Array.
          if (is_satisfactory(B))
          then (U, B)
          else Pot(A, ortho(mmmult(AU, transform(B)), diagonal(B)))
        end (mmmult(transpose(U), AU))
      end (mmmult(A, U))
    end
  end
end
```

(iii) **Unfolding and simplification**

```
fun Pot:real Array = ...
 λ AU:real Array.
  λ B:real Array. ...
  end (generate([n, n], λ [i, j].reduce([n], λ [k].U[k, i]*AU[k, j], plus, 0.0)))
 end (generate([n, n], λ [i, j].reduce([n], λ [k].A[i, k]*U[k, j], plus, 0.0)))
 ...
```

Applications of functions such as **mmmult** and **transpose** have been replaced by their definitions expressed as *generate* and *reduce* operations (see Section 2.2).

(iv) **Array form**

by *generate-reduce* rule xiii →

```
fun Pot:real Array = ...
 λ AU:real Array.
  λ B:real Array....
  end (reduce([n], λ [k].generate([n, n], λ [i, j]. U[k, i]*AU[k, j]), plus, 0.0))
 end (reduce([n], λ [k].generate([n, n], λ [i, j].A[i, k]*U[k, j]), plus, 0.0))
```

```
...

by Element Operator to Array Operator rule i →
fun Pot:real Array = ...
 λ AU:real Array.
  λ B:real Array....
  end (reduce([n], λ [k].generate([n, n], λ [i, j].U[k, i])
            *generate([n, n], λ [i, j].AU[k, j]),
        plus, 0.0))
 end (reduce([n], λ [k].generate([n, n], λ [i, j].A[i, k])
            *generate([n, n], λ [i, j].U[k, j]),
        plus, 0.0))
 ...

by Expand Special Case rule vii →
fun Pot:real Array = ...
 λ AU:real Array.
  λ B:real Array....
  end (reduce([n], λ [k].expand_cols([n], generate([n], λ [i].U[k, i]))
            * expand_rows([n], generate([n], λ [j].AU[k, j]),
        plus, 0.0)))
 end (reduce([n],λ [k].expand_cols([n], generate([n], λ [i].A[i, k]))
            * expand_rows([n], generate([n], λ [j].U[k, j]),
        plus, 0.0)))
 ...

by extraction rule viii →
fun Pot:real Array = ...
 λ AU:real Array.
  λ B:real Array....
  end (reduce([n], λ [k]. expand_cols([n], row(U, k))
            * expand_rows([n], row(AU, k),
        plus, 0.0)))
 end (reduce([n], λ [k]. expand_cols([n], column(A, k))
            * expand_rows([n], row(U, k),
        plus, 0.0)))
 ...
```

(v) **Common sub-expression elimination**

Common sub-expression elimination (CSE) has no effect on the example fragment from POT. Although there is a common element, **row(U, k)**, in the two matrix products it is not efficient to make this a common computation in this context. In fact, this operation is implemented using a particular form of DAP addressing so no computation need be performed to create a column of the array U.

(vi) **Functional form to imperative form**

```
subroutine Pot
...
  block
    real AU(n, n)
    do k=1, n, 1
      AU = AU+expand_cols([n], column(A, k)) *
                expand_rows([n], row(U, k))
    enddo
    block
    real B(n, n)
    do k=1, n, 1
      B = B+expand_cols([n], row(U, k)) *
                expand_rows([n], row(AU, k))
    enddo
    ...
    end
  end
end
```

The **reduce** operations are translated into loops over the index range.

(vii) **Array operations to DAP operations**

The abstract array operations are converted to the particular (and somewhat arcane) syntax required by Fortran Plus Enhanced.

```
subroutine Pot
...
  block
    real AU(*n, *n)
    do k=1, n, 1
      AU = AU+matc(A( , k), n)*matr(U(k, ), n)
    enddo
    block
    real B(*n, *n)
    do k=1, n, 1
      B = B+matc(U(k, ), n)*matr(AU(k, ), n)
    enddo
    ...
    end
  end
end
```

(viii) **Fortran standardization**

The main section of the Fortran Plus Enhanced implementation of POT is shown in Fig. 8.

```
      parameter(tol=1e-15, n=??)
      real A(*n, *n), U(*n, *n), AU(*n, *n)
      integer step, signD(*n, *n)
      real B(*n, *n), diagB(*n), D(*n, *n), g598(*n)
      logical mask(*n, *n), g650(*n, *n), g595(*n, *n)
      ....
100   continue
      AU=0
      do 110 step=1, n
      AU=AU+matc(A(, step), n)*matr(U(step, ), n)
110   continue
      B=0
      do 120 step=1, n
      B=B+matc(U(step, ), n)*matr(AU(step, ), n)
120   continue
      diagB=B(patunitdiag(n), )
      g598=abs(diagB)
      if ((sum(abs(B))-sum(g598))/(n*(n-1)).lt.tol) goto 200
      g595=patlowertri(n).and. .not. patunitdiag(n)
      mask=patlowertri(n)
      g650=patunitdiag(n)
      U(mask.and.g650)=1
      mask(g650)=.false.
      g650=g595
      D=matr(diagB, n)-matc(diagB, n)
      signD=1
      signD(D.lt.0)=-1
      U(mask.and.g650)=(-2*B)/(D+signD*sqrt(D*D+4*(B*B)))
126   U(.not.patlowertri(n))=-tran(S)
      D=0
      do 130 step=1, n
      D=D+matc(AU(step, ), n)*matr(S(, step), n)
130   continue
      S=R
      ....
      goto 100
200   continue
      ....
```

Fig. 8. Fortran Plus Enhanced implementation of POT.

- The * in the declaration of the matrices indicates that their elements are to be processed in parallel.
- The iteration of **POTiters** has been realized by a **GOTO** loop beginning at line 100 and ending at line 200.

- The loop which terminates at line 110 computes the product of matrices A and U. This product is stored since it is used twice: in the computation of B ($U^{\mathrm{T}} * A * U$) and in the new eigenvector matrix approximation U ($A * U * transform(B)$).
- The computation of B (the diagonal of which gives the current approximation to the eigenvalues) is completed at line 120.
- If B is sufficiently close to being diagonal (the mean of the absolute values of the off-diagonal elements is sufficiently close to zero) the loop is exited via the **GOTO 200** statement.
- The following lines, up to line 126, construct the transformation matrix. The definition of *transform* explicitly distinguishes elements in the lower triangle of the transformation matrix from elements in its upper triangle; its implementation on a SIMD architecture thus requires the computation of two matrices (one for lower triangle, one for upper triangle) which are then 'merged'. However, because the transformation matrix is (ignoring its diagonal) anti-symmetric, only one of these matrices need be computed (say, the matrix for the lower triangle); the other matrix can be formed by transposition and negation (as in line 126).

  Some of the mask manipulation in this part of the computation is unnecessary: no effort has been made to optimize mask expressions since they are very cheap on the DAP. (The grid of single-bit processing elements can manipulate the single-bit representation used for booleans very efficiently.)
- The eigenvector approximation matrix U is updated by the loop terminating at line 130. The orthonormalization of the columns of U is not shown.

The FPE implementation of POT is considerably different from its ML specification: the details of the computation of the matrix products and of the transformation matrix would be inaccessible to one unfamiliar with the DAP. The program is efficient but it is not easy to read. Of course, it is not intended that the FPE implementation *should* be read – it is nothing more than a source for processing by the FPE compiler to produce efficient machine code for the AMT DAP.

6.2. Conjugate gradient

The Fortran Plus Enhanced implementation of CG is shown in Fig. 9.

- The collection of vectors manipulated by the algorithm is realized by four arrays **x(*n)**, etc. The computation of the vectors from which the next approximation is constructed is performed using destructive updates on these arrays; thus there are no *separate* variables corresponding to **x'**, etc.
- The repetition required by the algorithm (expressed using **iterate**) is implemented using a loop realized by a **GOTO** occurring at line 13; the loop ends at line 15.
- Line 2 computes the inner product of **r** with itself. This value is the measure of the accuracy of the approximation to the solution.
- If the approximation is sufficiently accurate, the loop is exited via the **GOTO** statement on line 4.
- Otherwise, the next set of values (**x'**, **r'**, **p'** and **q'**) is computed by lines 6 to 12.

```
real*8 A(*n, *n), x(*n), r(*n), p(*n)
real*8 q(*n), b(*n), r1(*n), beta, rr
integer cnt

        ....
1 continue
2 rr = sum(r*r)

3 if (sqrt(rr).lt. 1.0E-14) then
4 goto 15
5 else
6 alpha = rr/sum(q*q)
7 r1 = r-sumr(A*matc(q, n))*alpha
8 beta = sum(r1*r1)/rr
9 q = sumc(A*matr(r1, n))+q*beta
10 x = x+p*b
11 r = r1
12 p = r1+p*beta
13 goto 1
14 endif
15 continue
        ....
```

Fig. 9. Fortran Plus Enhanced implementation of conjugate gradient.

Note, in particular, that lines 7 and 9 compute the two matrix–vector products:

transpose(A)*q → sumr(A*matc(q, n))
A*r' → sumc(A*matr(r1, n))

In the first product, the matrix A is transposed, but no explicit transpose operation occurs in the implementation; rather, row and column operations in the implementation of normal matrix–vector multiplication (i.e. without transposition) are interchanged. This accounts for the slight difference in form between the implementations of the two products.

Again, the DAP implementation may appear rather ugly since it is not intended for a human reader. The program is, however, an extremely efficient implementation that exploits the strengths (and indeed quirks) of the DAP architecture. The implementation makes effective use of the DAP hardware, with all of the vector and matrix operations being performed in fully data-parallel manner. The only unsatisfactory aspect of the implementation is the unnecessary use of the variable **r1**: the assignment to **r1** in line 11 could be replaced with an assignment to **r**, obviating the need to assign to **r** later. Efficiency could be improved by eliminating two vector assignments and one vector variable.

Matrix Size	*Time per iteration (sec)*	
	Hand Crafted Fortran Plus Enhanced	Automatically Derived Fortran Plus Enhanced
64	1.35	1.35
128	9.30	9.31
256	69.86	70.30

Fig. 10. Execution times of derived and manually constructed DAP implementations of POT.

7. Execution performance of derived implementations

From the point of view of a user of an implementation, its most important feature (after correctness) is its execution speed. Clear, extensible functional specifications are useful *only* if it is possible to derive fast and efficient implementations from them.

Examination of the derived implementations reveals that they are highly efficient – they make excellent use of the parallel processing capabilities of the DAP. A more rigorous assessment of the execution performance of the derived POT implementation can be made by comparing it with that of a hand-crafted implementation developed independently by a programmer who was very familiar with the target architecture.

In Fig. 10 the time required to compute one approximation to the eigensystem[7] by a hand-crafted implementation of POT is compared with the time required by the automatically derived implementation; the hand-crafted version has been analysed in [21, 70].

The execution times for the hand-crafted and automatically derived versions are almost identical. For the larger matrix examples the derived implementation is marginally slower than the hand crafted version (by between 0.1% and 0.6%). This discrepancy arises from a minor optimization made possible by the particular way in which the hand-crafted version implements the *transform* operation.

8. Related work

The work presented in this paper addresses many different themes in computing science. Thus it is impossible to provide an exhaustive survey of related work. However, the work described here treats the broad themes of language selection for algorithm specification, functional language compilation and program transformation.

[7] The same amount of time is required to generate each successive approximation.

8.1. Algorithm specification

The primitive array functions, *generate* and *reduce* (see Section 2) used in describing computations should be familiar to those with experience of functional programming languages. The definitions presented are, for the most part, natural extensions of the usual definitions over lists to definitions over arrays. (Those unfamiliar with functional programming languages may consult [44, 7, 30, 63, 67, 26] for an introduction to functional programming and the use of higher-order functions.) No claim is made in respect to the originality of the array functions; they are presented as objects that have proved to be particularly useful in the specification of numerical mathematical algorithms and in the formal manipulation of such specifications.

Maaßen [50] proposes data structures and higher-order functions over them for the parallel execution of functional programs. The functions employed in the functional specifications in this paper are related to these definitions.

Darlington et al. [29] use *skeletons* [28, 23, 61] in high-level specifications of algorithms. Skeletons are higher-order functions that describe a repertoire of parallel operations and are used as the building blocks of an algorithm's specification. Skeletons are intended to separate the meaning of the computation from any tailored parallel architecture form which may be derived from such definitions. The primitive functional forms used here may be regarded as simple skeletons in that they may be interpreted as indicating data-parallel execution.

In [28, 29] skeletons that are oriented to particular computational models are outlined; for example, processor-pipeline and processor-farm skeletons are defined. This type of skeleton may be viewed as defining an execution model which is suitable for carrying out a particular computation. This approach to algorithm specification is different from the one adopted here; in this paper, it is proposed that a specification should be as free from execution detail as possible – the algorithm specification defines only the functions to be implemented and relegates the decisions as to implementation to the transformation phase. It is clear that automatic tools (such as the transformation system suggested here) could not supersede the role of the expert programmer; nevertheless, it is interesting to explore how much can be achieved automatically. With TAMPR it is possible to apply particular algorithm transformations to achieve the effect of model-oriented skeletons.

The Bird Meertens Formalism (BMF) [65, 5, 6, 56, 3, 52] provides a simple, consistent functional language in which algorithms may be expressed. BMF provides an elegant framework for the study of algorithms, but its utility as a numerical mathematical algorithm specification language is problematical given its list-based approach. The array is fundamental to the natural expression of a large body of numerical mathematical algorithms and to their efficient implementation. We contend that, for most numerical mathematical algorithms a functional specification that uses lists to represent arrays is unnatural – for example, consider expressing a basic operation such as matrix transpose

using a list representation. Moreover, such list-based specifications are unlikely to be amenable to the utilization of the optimization techniques and implementation strategies developed by implementers of numerical mathematical algorithms. This corpus of implementation experience is essential for efficient implementation of functional, numerical mathematical specifications and thus for the acceptance of functional programming languages for this purpose. Numerical mathematicians readily accept array-based functions as a natural extension to the conventional mathematical notations used in their community.

Hains and Mullin [36] define ML functions that operate on arrays. The dimensionality of the array is expressed by defining the structure of the array. However, as with BMF, arrays are represented by lists of elements thereby reducing the readability of specifications and impairing its usability for those to whom the work reported here is particularly addressed.

8.2. Functional language compilers

Many functional language compilers generate machine code which is comparable in efficiency to that produced from hand-crafted imperative programs; among these are the Orbit Compiler [49] for the language T, the ALFL language compiler [8], the compiler for the SISAL language [26] and the Lazy ML compiler [2]. This body of experience has been drawn upon in the compiler-oriented transformations of the transformational derivations presented here.

Many computer systems have been developed specifically to support the parallel execution of functional programming languages [24, 40, 45, 64, 48, 38, 53]. Special hardware that supports combinatoric graph reduction offers the possibility of a radical change in the relative performances of functional and imperative languages, thereby reducing the need for the construction of an imperative implementation of a functional specification. Simon Peyton-Jones [63] gives an excellent survey and description of combinatoric graph reduction. Although attractive in principle, very few special-purpose graph reduction machines have been constructed and none is widely available. Even if a successful graph-reduction machine were built and could yield execution performance comparable to that achieved by procedural programs executed on conventional Von Neumann architectures, such a machine is unlikely to be a cost-effective alternative to mass-produced conventional machines.

A number of functional languages have been extended to include parallel evaluation primitives. Typically, when using such languages, a programmer specifies that a process be created to evaluate some expression and evaluation then proceeds until the value generated by the created process is required [33, 55, 37, 49, 34]. Such language forms might serve as a target for transformation derivation or as a standard form to be used in the transformational process. As before, however, our goal is to have specifications that are free of execution detail.

8.3. Program transformation

A large volume of literature on program transformations and derivations is available. Although it is not the main subject of this paper the interested reader is referred to Partsch [60] for an overview of various transformations systems and to [25, 27, 72, 66, 42, 47] for discussions of particular transformation systems.

A major issue still to be addressed in transformation systems is the control of the derivation process; i.e. the specification of *strategies* to achieve some goal. The approach advocated here is to define a sequence of *normal forms* that achieves a goal (the conversion from some initial form to some final form): consideration of strategy is then reduced to ensuring that the transformations convert one normal form into the next. The use of normal forms has been discussed at least as early as 1970 by Boyle [10] and has been addressed more recently by Hoare [43]. In a recent paper, Boyle [17] shows how a sequence of normal forms can be used to control transformations that perform partial evaluation of programs.

Program transformation has traditionally been used to recast a program into an equivalent but more efficient form. The initial and final forms are generally expressed in the same language. An early example is Burstall's and Darlington's unfold–fold transformations [18] which improve the execution efficiency of systems of recursive equations. This topic is pursued further in [9, 39, 46, 62]. Again, the work reported in these papers has been employed in the optimization techniques used in the unfolding phase of the transformational method discussed here.

8.4. Traditional imperative parallel programming

The majority of programs that are executed by parallel computers are expressed in Fortran, a language which is inherently sequential. Fortran compilers which generate code for parallel systems usually perform extensive program analysis in order to exploit parallel execution. This is achieved primarily by executing multiple iterations of **DO** loops simultaneously [73]. This area of study is not *directly* related to the work in this paper, insofar as the results of research in this area are not employed in the derivations presented here. However, the research *is* important because Fortran is currently the only feasible language for programming many vector and parallel computer systems (and that has consequences for derivations). The intractability of many of the problems that arise in vectorization or parallelization is a major factor motivating research into alternative approaches to programming high-performance computer systems; the work reported here may be viewed as one alternative.

Configuration languages such as those advocated in [31, 51] permit composition of *black-box* processes by specification of the communication between these. Typically, the processes are expressed in a sequential language, such as Fortran or C, and the communication is by reading and writing to communication ports. Configuration languages normally require too low a level of detail to be suitable for specifying algorithms, but they might be suitable as target languages for derivations.

Imperative languages have been extended to include parallel programming constructs. The extensions range from subroutine libraries, that are little more than interfaces to operating system routines, to entirely new languages such as ADA, which are designed with parallel execution in mind. Of particular relevance in the context of this paper are the array extensions to Fortran provided by languages such as Fortran90 [57], Connection Machine Fortran [22], Fortran Plus Enhanced [1], Fortran-D [32] and Vienna Fortran [4]. These extensions provide, to some degree, a data abstraction for arrays: many common operations such as the elementwise addition of two arrays are provided as pure functions (denoted by the usual "+" operator). Vienna Fortran is distinguished from the others by its advanced support for *data templates*, which permit the programmer to define the distribution of data on distributed memory systems. Recently, many of the features of these array-based Fortran dialects have been coalesced into a single language called High Performance Fortran [41]. The language definition is still under review and there are, as yet, no widely available HPF compilers.

In some ways, the array extensions to Fortran may be viewed as an attempt to introduce into Fortran some of the features of functional languages: expressions permit array operations to be denoted in a high-level, machine-independent manner that allows operations to be succinctly combined and that facilitates analysis.

It is thus natural to enquire whether the wide spread use of array-based Fortran would render irrelevant the work reported in this paper, since programmers would have available array operations that are almost the same as those provided by the array data abstraction used here. We offer the following reasons for replying in the negative:

- The array-based Fortran dialects fall short of providing complete data abstractions for matrices and vectors; for example, they do not support common linear algebraic operations such as matrix product.
- Some of the Fortran dialects *do* provide module mechanisms for hiding implementation details, but, in general, efficiency considerations will probably force programmers to continue using subroutines as their main (if any) decomposition mechanism. What the derivational approach offers over Fortran in any form is a clear separation of the tasks of specifying an algorithm and implementing it.
- The expression-based array operations are likely to impose just as high overheads on Fortran implementations as on functional implementations. The developers of compilers for the Fortran dialects will have to address issues such as destructively updating arrays, but they will have to address the issue in the context of an already complex compilation system. The derivational approach allows implementation issues to be separated and addressed more methodically.

Thus, the chief relevance of the array-based dialects of Fortran for the derivational approach proposed here will probably result from their use as programming models to replace the ill-defined model provided by Fortran77. (It should be easier to derive implementations designed for parallel execution using HPF as the implementation language rather than Fortran77.)

9. Conclusion

In this paper we demonstrated that it is possible to transform mechanically high-level functional specifications into highly efficient implementations tailored for execution on the AMT DAP array processor. The functional specifications are not biased in ways that guarantee efficiency of their implementations on a particular machine architecture; rather, they are expressed in ways that provide clear statements of algorithms. Indeed, the example specifications may be used as starting points for deriving similarly efficient implementations tailored for execution on other machines.

The transformations used to produce the implementations presented in this paper are problem *independent* and may be applied to ML specifications of other algorithms. The method may be further refined by tailoring the generated code for a particular compiler (for example, producing sectioned Fortran Plus array operations that are tailored for the size of the processor array) and defining specialized data transformations (for example, specific transformations for sparse matrices).

References

[1] AMT, *Fortran-Plus Language Enhanced*, AMT Ltd., 1988.
[2] L. Augustsson and T. Johnsson, The Chalmers lazy-ML compiler, *Comput. J.* **32** (2) (1989) 127–141.
[3] R. Backhouse, An exploration of the Bird–Meertens formalism, Technical Report CS8810, Department of Mathematics and Computing Science, University of Groningen, 1988.
[4] S. Benker et al., Vienna Fortran-90, in: R. Voigt and J. Saltz, eds., *Proc. Scalable High-Performance Computing Conf.* (IEEE Computer Society Press, Silver Spring, MD, 1992) 51–59.
[5] R.S. Bird, Algebraic identities for program calculation, *Comput. J.* **32** (2) (1989) 122–126.
[6] R.S. Bird and J. Hughes, The Alpha-Beta algorithm: An exercise in program transformation, *Inform. Process. Lett.* **24** (1987) 53–57.
[7] R.S. Bird and P. Wadler, *Introduction to Functional Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1988).
[8] A. Bloss, P. Hudak and J. Young, An optimizing compiler for a modern functional language, *Comput. J.* **32** (2) (1989) 152–161.
[9] E.A. Boiten, Improving recursive functions by inverting the order of evaluation, *Sci. Comput. Programming* **18** (1992) 139–179.
[10] J.M. Boyle, A transformational component for programming language grammar, ANL-7690, Argonne National Laboratory, July 1970.
[11] J.M. Boyle, Program adaptation and program transformation, in: R. Ebert, J. Lueger and L. Goecke, eds., *Practice in Software Adaptation and Maintenance* (North-Holland, Amsterdam, 1980) 3–20.
[12] J.M. Boyle and M.N. Muralidharan, Program reusability through program transformation, *IEEE Trans. Software Engrg.* **SE-10** (5) (1984) 574–588.
[13] J.M. Boyle and T.J. Harmer, Functional specifications for mathematical computations, in: B. Möller, ed., *Proc. IFIP TC2/WG2.1 Working Conf. on Constructing Programs from Specifications*, Pacific Grove, CA, 13–16 May 1991 (North-Holland, Amsterdam, 1991) 205–224.
[14] J.M. Boyle, Abstract programming and program transformations – An approach to reusing programs, in: T.J. Biggerstaff and A.J. Perlis, eds., *Software Reusability, Vol. I* (ACM Press (Addison-Wesley), New York, NY, 1989) 361–413.
[15] J.M. Boyle and T.J. Harmer, A practical functional program for the CRAY X-MP, *J. Funct. Programming* **2** (1992) 81–126.
[16] J.M. Boyle, Towards automatic synthesis of linear algebra programs, in: M.A. Hennell and L.M. Delves, eds., *Production and Assessment of Numerical Software* (Academic Press, New York, 1980) 223–245.

[17] J.M. Boyle, Automatic, self-adaptive control of unfold transformations, Presented at *PROCOMET '94, IFIP Working Conf. on Programming Concepts, Methods and Calculi*, San Miniato, Italy, 6–10 June 1994, Proceedings to be published by North-Holland.

[18] R.M. Burstall and J. Darlington, A transformation system for developing recursive programs, *ACM J.* **24** (1977) 44–67.

[19] M. Clint et al., Towards the construction of an eigenvalue engine, *Parallel Comput.* **8** (1988) 127–132.

[20] M. Clint et al., A comparison of two parallel algorithms for the symmetric eigenproblem, *Internat. J. Comput. Math.* **15** (1984) 291–302.

[21] M. Clint, J.S. Weston and C.W. Bleakney, Comparison of parallel Fortran environments on the AMT DAP 510 for a linear algebra application, *Concurrency: Practice and Experience* **6** (1994) 193–204.

[22] *CM Fortran Reference Manual*, TMC, Thinking Machines Corporation, Cambridge, MA, 1991.

[23] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation* (MIT Press, Cambridge, MA, 1989).

[24] A. Contessa et al., MaRS, a combinator graph reduction multiprocessor, in: E. Odijk, M. Rem and J.C. Syre, eds., *PARLE '89 Parallel Architectures and Languages Europe*, Vol. I, Lecture Notes in Computer Science, Vol. 365 (Springer, Berlin, 1989) 176–192.

[25] M.S. Feather, A system for assisting program transformation, *ACM Programming Languages* **4** (1982) 1–20.

[26] J.T. Feo, D. Cann and R.R. Oldehoeft, A report on the SISAL language project, *J. Parallel Distributed Comput.* **10** (4) (1990) 349–366.

[27] J. Darlington et al., A functional programming environment supporting execution, partial execution and transformation, in: E. Odijk, M. Rem and J.C. Syre, eds., *PARLE '89 Parallel Architectures and Languages Europe, I*, Lecture Notes in Computer Science, Vol. 365 (Springer, Berlin, 1989) 286–305.

[28] J. Darlington et al., Parallel programming using skeletons, in: A. Bode, M. Reeve and G. Wolf, eds., *PARLE'93: Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science, Vol. 694 (Springer, Berlin, 1993).

[29] J. Darlington, Yi-ke Guo and Hing Wing To, Structured parallel programming: Theory meets practice, Draft paper obtained from the first author, January 1995.

[30] A.J. Field and P.G. Harrison, *Functional Programming* (Addison-Wesley, Reading, MA, 1988).

[31] I. Foster et al., Productive parallel programming: The PCN approach, *Scientific Programming* **1** (1) (1992) 51–66.

[32] G. Fox et al., Fortran D language specification, Research Report, Rice University, January 1992.

[33] R. Gabriel and J. McCarthy, QLISP, in: *Parallel Computation and Computers for AI* (Kluwer Academic Publishers, Dordrecht, 1988) 63–89.

[34] A. Giacalone et al., Facile: a symmetric integration of concurrent and functional programming, *J. Parallel Comput.* **18** (2) (1989) 121–160.

[35] A. Gill et al., A short cut to deforestation, in: *FPCA '93, Conf. on Functional Programming Languages and Computer Architecture* (ACM Press, New York, 1993) 223–232.

[36] G. Hains and L.M.R. Mullin, Parallel functional programming with arrays, *Comput. J.* **36** (3) (1993) 238–245.

[37] R.H. Halstead, Parallel computing using multilisp, in: J.S. Kowalik, eds., *Parallel Computation and Computers for AI* (Kluwer Academic Publishers, Dordrecht, 1987) 21–40.

[38] P.G. Harrison and M. Reeve, The parallel graph reductions machine ALICE, in: J.H. Fasel and R.M. Keller, eds., *Graph Reduction*, Lecture Notes in Computer Science, Vol. 279 (Springer, Berlin, 1986) 181–202.

[39] P.G. Harrison and H. Khoshnevisan, Algebraic transformation techniques for functional languages, *Comput. J.* **31** (1988) 229–242.

[40] L.O. Hertzberger and W.G. Vree, A coarse grain parallel architecture for functional languages, in: E. Odijk, M. Rem and J.C. Syre, eds., *PARLE '89 Parallel Architectures and Languages Europe, I*, Lecture Notes in Computer Science, Vol. 365 (Springer, Berlin, 1989) 269–285.

[41] *High Performance Fortran Language Specification*, Version 1.1, High Performance Fortran Forum, November 1994.

[42] D. Hildum and J. Cohen, A language for specifying program transformations, *IEEE Trans. Software Engrg.* **16** (6) (1990) 630–638.

[43] C.A.R. Hoare et al., Normal form approach to compiler design, *Acta Inform.* **30** (8) (1993) 701–739.
[44] J. Hughes, Why functional programming matters, *Computer J.* **32** (2) (1989) 98.
[45] S. Hwang and D. Rushall, The nu-STG machine: a parallelized spineless tagless graph reduction machine in a distributed memory architecture, in: *Proc. 4th Internat. Workshop on the Parallel Implementation of Functional Languages*, 1992.
[46] O. Kaser et al., On the conversion of indirect to direct recursion, *ACM Lett. Programming Languages Systems* **2** (1993) 151–164.
[47] E.W. Karlsen et al., The PROSPECTRA system: A unified development framework, in: M. Nivat, C. Rattray, T. Rus and G. Scollo, eds., *Algebraic Methodology and Software Technology (AMAST '91)* (Springer, Berlin, 1991) 421–433.
[48] J.A. Keane, An overview of the flagship system, *J. Funct. Programming* **4** (1) (1994) 19–45.
[49] D.A. Krantz, R.H. Halstead and E. Mohr, MulT: a high-performance parallel Lisp, *ACM SIGPLAN Notices* **24** (7) (1989) 81–90.
[50] A. Maaßen, Parallel programming with data structures and higher order functions, *Sci. Comput. Programming* (1992) 1–38.
[51] J. Magee et al., An overview of the REX software architecture, *2nd IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, 1990.
[52] G. Malcolm, Data structures and program transformations, *Sci. Comput. Programming* **14** (1990) 255–279.
[53] Shogo Matsui et al., SYNAPSE: a multi-microprocessor lisp machine with parallel garbage collector, in: *Parallel Algorithms and Architectures*, Lecture Notes in Computer Science, Vol. 269 (Springer, Berlin, 1987) 131–137.
[54] C. McCrosky, Intermediate container removal, *Comput. Languages* **16** (2) (1991) 179–195.
[55] P.F. McGehearty and E.J. Krall, Potentials for parallel execution of common lisp programs, in: K. Hwang, S.M. Jacobs and E.E. Swartzlander, eds., *Proc. 1986 Internat. Conf. on Parallel Processing* (Computer Society Press, 1986) 696–702.
[56] L. Meertens, Constructing a calculus of programs, in: *Mathematics of Program Construction*, Lecture Notes in Computer Science, Vol. 375 (Springer, Berlin, 1989) 66–90.
[57] M. Metcalf and J. Reid, *Fortran 90 Explained* (Oxford University Press, Oxford Science Publications, 1990).
[58] J.J. Modi, *Parallel Algorithms and Matrix Computations* (Oxford University Press, Oxford, 1988).
[59] D. Parkinson and J. Litt, eds., *Massively Parallel Computing with the DAP*, Research Monographs in Parallel and Distributed Computing (MIT Press, Cambridge, MA, 1990).
[60] H. Partsch and R. Steinbrüggen, Program transformation systems, *ACM Comput. Surveys* **15** (3) (1983) 199–236.
[61] S. Pelagatti, A methodology for the development and support of massively parallel programs, Ph.D. Thesis, Universita Delgi Studi di Pisa, 1993.
[62] A. Pettorossi and R.M. Burstall, Deriving very efficient algorithms for evaluating linear recurrence relations using the program transformation technique, *Acta Inform.* **18** (1982) 181–206.
[63] S.L. Peyton-Jones, *The Implementation of Functional Programming Languages* (Prentice-Hall, New York, 1987).
[64] S.L. Peyton Jones et al., High-performance parallel graph reduction, in: E. Odijk and J.-C. Syre, eds., *PARLE Parallel Architectures and Languages Europe, I*, Lecture Notes in Computer Science, Vol. 365 (Springer, Berlin, 1989) 193–206.
[65] D.B. Skillicorn, Architecture-independent parallel computation, *IEEE Comput.* **23** (12) (1990) 38–50.
[66] D.R. Smith, KIDS: A semiautomatic program development system, *IEEE Trans. Software Engrg.* **16** (9) (1990) 1024–1043.
[67] G.L. Steele, *Common Lisp* (Digital Press, Belford, MA, 1986).
[68] P. Wadler, Deforestation: transforming programs to eliminate trees, *J. Theoret. Comput. Sci.* **73** (1990) 231–248.
[69] A. Wilström, *Functional Programming using Standard ML* (Prentice-Hall, London, 1987).
[70] J. Weston and M. Clint, Two algorithms for the parallel computation of eigenvalues and eigenvectors of large symmetric matrices using the ICL DAP, *Parallel Comput.* **13** (1990) 281–288.

[71] J. Weston et al., The parallel computation of eigenvalues and eigenvectors of large Hermitian matrices using the AMT DAP 510, *Concurrency: Practice and Experience* **3** (3) (1991) 179–185.

[72] J.A. Yang and Young-il Choo, Parallel-program transformation using a metalanguage, *ACM SIGPLAN Notices* **26** (7) (1991) 11–20.

[73] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, Frontier Series (ACM, New York, 1990).